Stacks

CMPUT 115 - Lecture 21
Department of Computing Science
University of Alberta

Some code in this lecture is based on code from the book
Java Structures by Duane A. Bailey or the companion structure package.

About This Lecture

- In this lecture we will a special linear structure called a Stack.

Outline

- Stack Interface
- Maze - An example of using a Stack
- Vector Implementation of Stack
- List Implementation of Stack

Stack

- A list with the following unique properties
  - A list
  - Last in, first out

Structure Interface - Stack (slide 1)

```java
public interface Stack extends Linear {
    public void push(Object anObject);
    // post: the object that is added will be popped
    // next if nothing else is pushed.

    public Object pop();
    // pre: stack is not empty
    // post: removes the most recently pushed object and
    // returns it. Same operation as remove

    public Object peek();
    // pre: stack is not empty
    // post: returns the next object to be removed, but does
    // not remove it
}
```

Structure Interface - Stack (slide 2)
Stack Example - Stack frames

- One common application of Stacks is to hold the program state during the execution of a method (or procedure or function).
- In this case, the objects on the Stack are stack frames.
- In recursive calls, many stack frames can be created in the virtual machine.
- It is possible to simulate recursion with a loop and a stack to hold all the values that would normally be stored in the stack frame created by a recursive call.

Stack Example - Maze Algorithm

- One common approach to solving search problems is to use a Stack to hold unsearched paths.
- Select the start square as the current square.
- Repeat as long as the current square is not null and is not the finish square:
  - “Visit” the square and mark it as visited.
  - Push one square on the stack for each unvisited legal move from the current square.
  - Pop the stack into the current square or bind the current square to null if the stack is empty.
- If the current square is the goal we are successful, otherwise there is no solution

Maze Program (slide 1)

```java
public static void main(String[] args) {
    // A Maze program
    Maze maze;
    Position goal;
    Position current;
    Linear todo;
    maze = new Maze(args[0]); // Create a maze from a file
    goal = maze.finish();
    todo = new StackList(); // A class that implements Stack
    current = maze.start();
    maze.visit(current);
    todo.add(current);
    while ((current != null) && (!current.equals(goal))) {
        maze.visit(current);
        pushUnvisitedNeighbor(todo, maze, current.north());
        pushUnvisitedNeighbor(todo, maze, current.south());
        pushUnvisitedNeighbor(todo, maze, current.east());
        if (todo.isEmpty())
            current = null;
        else
            current = (Position) todo.remove();
    }
    if (current == null)
        System.out.println("No solution exists");
    else
        System.out.println("Solution found");
    System.out.println(maze);
}
```

Maze Program (slide 2)

```java
public static void pushUnvisitedNeighbor (Linear todo, Maze maze, Position position) {
    // Pre: position != null
    // Post: If the given position is unvisited and is clear (not a wall) in the given maze, it is pushed onto the given Stack.
    if (maze.isClear(position) && (!maze.isVisited(position)))
        todo.add(position);
}
```

Maze Program (slide 3)
Vector-Based Stacks

- We can implement a Stack using a single Vector.
- This implementation is space and time efficient.
- It is also simple, so few mistakes can be made.
- However, the time for adding an element can vary, depending on whether the Vector must grow or not.
- Since the Vector will never shrink (in our implementation), the physical memory required is at least as large as the maximum Stack size, even if the Stack shrinks.

StackVector

- How to store the data?
  - A vector
- Constructor?
  - to construct a vector with the given size
- Three major methods
  - push
  - pop
  - peek

StackVector - State and Constructors

```java
public class StackVector implements Stack {
    /*
     * An implementation of the Stack Interface that uses
     * a single Vector to store the Stack elements.
     */
    // Instance Variables
    protected Vector data;
}
```

StackVector – Implementation of Linear Interface

```java
public void push(Object anObject) {
    // post: the object that is added will be popped next if
    // nothing else is added. Same operation as push
    this.data.addElement(anObject);
}
```

StackVector – Implementation of Linear Interface

```java
public Object pop() {
    // pre: stack is not empty
    // post: removes the most recently added object and
    // returns it. Same operation as pop
    Object result;
    result = this.data.elementAt(this.data.size() - 1);
    this.data.removeElementAt(this.data.size() - 1);
    return result;
}
```
StackVector - Implementation of Linear Interface contd.

```java
public Object peek() {
    // pre: stack is not empty
    // post: returns the next object to be removed, but does not remove it
    return this.data.elementAt(this.data.size() - 1);
}
```

Listing based on Bailey pg. 132

StackVector - Implementation of Store Interface

```java
public int size() {
    // post: returns the number of elements contained in the store.
    return this.data.size();
}

public boolean isEmpty() {
    // post: returns the true iff store is empty.
    return this.data.size() == 0;
}

public void clear() {
    // post: clears the store so that it contains no elements.
    this.data.clear();
}
```

Listing based on Bailey pg. 134

List-Based Stacks

- We can implement a Stack using a SinglyLinkedList.
- Recall that operations at the head of a SinglyLinkedList take constant time so this implementation is time efficient.
- As the Stack grows and shrinks, so does the list, unlike the Vector implementation that never shrinks.
- The time for adding a single element is always the same, unlike the Vector implementation
- There is extra link space required for each element.
- The implementation is simple, so few mistakes can be made.

StackList - State and Constructors

```java
public class StackList implements Stack {
    // An implementation of the Stack Interface that uses a SinglyLinkedList to store the Stack elements.
    // Instance Variables
    protected List data;

    public StackList() {
        // post: initialize the Stack to be empty
        this.data = new SinglyLinkedList();
    }
```

Listing based on Bailey pg. 134

StackList - Implementation of Linear Interface

```java
public void add(Object anObject) {
    // post: the object that is added will be popped next if nothing else is added. Same operation as push
    this.data.addToList(anObject);
}

public Object remove() {
    // pre: stack is not empty
    // post: removes the most recently added object and returns it. Same operation as pop
    return this.data.removeFromHead();
}
```

Listing based on Bailey pg. 135
StackList – Implementation of Stack Interface

```java
public void push(Object anObject) {
    // post: the object that is added will be popped next if
    // nothing else is pushed. Same operation as add
    this.add(anObject);
}

public Object pop() {
    // pre: stack is not empty
    // post: removes the most recently pushed object and
    // returns it. Same operation as remove
    return this.remove();
}
```

StackList – Implementation of Store Interface

```java
public int size() {
    // post: returns the number of elements contained in
    // the store.
    return this.data.size();
}

public boolean isEmpty() {
    // post: returns the true iff store is empty.
    return this.size() == 0;
}

public void clear() {
    // post: clears the store so that it contains no
    // elements.
    this.data.clear();
}